Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#36: Port Driver Specifications

Revised by: Matt Deatherage & Suki Lee — September 1989
Written by: Dan Hitchens — May 1988

This Technical Note describes how to write your own drivers for Apple IIGS ports.
Changed since January 1989: Added description of new port driver structure.

Introduction

A port driver handles certain hardware-specific duties for the Print Manager, such as initializing firmware and handling low-level hardware handshaking protocols, if any are implemented. The port driver structure, like the printer driver structure, insulates the Print Manager from low-level details of printers and interface cards (or ports) so that the same calls work across various hardware configurations, provided drivers are installed on the boot disk.

Note that a port driver could also easily be called a card driver; the term port is used because the first ones written were for the internal ports of the Apple IIGS. A port driver could interface any printer (for which there is a printer driver) with any kind of port or peripheral card that can handle it. A familiar example would be a parallel printer interface card—a port driver for a parallel card would enable the Print Manager to print graphics to any parallel printer connected to it (provided, again, there was a printer driver for the particular printer installed).

In general, you need a port driver for each port or interface card through which you intend to print, and a printer driver for each printer to which you intend to print. On System Disk 4.0, Apple provides port driver files for the printer port (PRINTER), the modem port (MODEM), a port connected to the AppleTalk network (APPLETALK), and a parallel printer interface card (PARALLEL.CARD). Apple also provides printer drivers for the ImageWriter and ImageWriter II (IMAGEWRITER), the ImageWriter LQ (IMAGEWRITER.LQ), the LaserWriter family.(LASERWRITER), and an Epson (EPSON). With this configuration, you can print to any of the printer types above through any of the ports, cards, or over AppleTalk. Other printer drivers and port drivers would extend the user's selection of available configurations.

What’s in a Port Driver

File Structure

Users can install new port drivers into the system by copying a port driver file into a subdirectory called DRIVERS within the SYSTEM subdirectory or by running the Installer if the driver is supplied with a script to install it. The port driver file must be of type $BB. There are two kinds of port drivers: local drivers, intended to drive a printer connected locally, and network drivers, which handle printers connected over an AppleTalk network. Local drivers have an auxiliary type of $0002, and AppleTalk drivers (there should be only one) have an auxiliary type of $0003.

Port Driver Calls

A port driver must support the following calls:

`PrDevPrChanged`	$1913	
`PrDevStartup`	$1A13	
`PrDevShutDown`	$1B13	
`PrDevOpen`	$1C13	
`PrDevRead`	$1D13	
`PrDevWrite`	$1E13	
`PrDevClose`	$1F13	
`PrDevStatus`	$2013	
`PrDevAsyncRead`	$2113	(alias `PrDevInitBack`)
`PrDevWriteBackground`	$2213	(alias `PrDevFillBack`)
`PrPortVer`	$2413	
`PrDevIsItSafe`	$3013	

Note that a network port driver has much more work to do than a regular (local) port or card driver. A local driver only has to worry about one printer, whereas a network port driver may find that there is not even a printer available on a running network. The information on network drivers is provided mostly for informational purposes; you should never find it necessary to write your own AppleTalk port driver.

Entering and Exiting a Port Driver

Entering and exiting is the same as described for the printer driver calls in Apple IIGS Technical Note #35, Printer Driver Specifications. The new driver structure described there applies as well. As of this writing, there are no optional calls a port driver may support. The documented list must be supported in its entirety.

PrDevPrChanged **$1913**

Description:
The Print Manager makes this call every time the user accepts this port driver in the Choose Printer dialog.

Input:	LONG	printer name pointer

Direct Connect:
- Makes sure that this port has been set up correctly in the Control Panel (parity, baud rate, etc.), and puts up an alert for the user if it has not been. Remember that if you change settings, even at the user's request, you should change the Battery RAM parameters as well, so the setting changes will be reflected when the user enters the Control Panel.

Network:
- Copies the printer name to local storage for use in the `NBPLookup` function of the AppleTalk `PAPopen` and `PAPstatus` calls, usually by placing it in the AppleTalk parameter block. This function is similar to that performed by `PrStartUp`, except that `PrDevPrChanged` is called whenever the printer is changed by the user with the Choose Printer dialog.

PrDevStartUp **$1A13**

Description:
This call is not required to do anything. However, if your driver needs to initialize itself by allocating memory or other setup tasks, this is the place to do it. Network drivers should copy the printer name to a local storage area for later use.

Input:	LONG	printer name pointer
	LONG	zone name pointer

Direct Connect:
- Required to do nothing. This is a good place to do your own set-up tasks, if you have any.

Network:
- Copies the printer name and the zone name to local storage for use in the `NBPLookup` function of the AppleTalk `PAPopen` and `PAPstatus` calls, usually by placing it in the AppleTalk parameter block.

PrDevShutDown **$1B13**

Description:
This call, like `PrDevStartUp`, is not required to do anything. However, if your driver performs other tasks when it starts, from the normal (allocating memory) to the obscure (installing heartbeat tasks), it should undo them here. If you allocate anything when you

start, you should deallocate it when you shutdown. Note that this call may be made without a balancing `PrDevStartUp`, so be prepared for this instance. For example, do not try to blindly deallocate a handle that your `PrDevStartUp` routine allocates and stores in local storage; if you have not called `PrDevStartUp`, there is no telling what will be in your local storage area.

Input:	none

PrDevOpen $1C13

Description:

This call basically prepares the firmware for printing. It must initialize the firmware for both input and output. Input is required so the connected printer may be polled for its status.

A network driver has considerably more work to do, including the possibility of asynchronous communications. Details are provided below.

Input:	LONG	completion routine pointer
	LONG	reserved long

Direct Connect:

- Initializes the firmware for input and output, preparing for reading from or writing to the printer.
- If the completion pointer is `NIL`, then `RTL`. If it is not `NIL`, then perform a `JSL` to the completion routine.

Network:

- Initializes the `End-Of-Write` parameter in the AppleTalk `PAPWrite` parameter block to zero. Never call AppleTalk `INIT` to initialize the firmware.
- If the completion pointer is `NIL`, then prepares for synchronous communications. If it is not `NIL`, prepares for asynchronous printing.
- Calls AppleTalk `PAPopen` to make connection, returning an error if one is returned to you.
- Stores the AppleTalk Session number in the `PAPRead`, `PAPWrite` and `PAPClose` parameter blocks.
- Executes an `RTL` if there is no completion routine (pointer is `NIL`), otherwise perform a `JSL` to the completion routine.

PrDevRead $1D13

Description:

This call reads input from the printer.

Input:	WORD	space for result
	LONG	buffer pointer
	WORD	number of bytes to transfer

Output:	WORD	number of bytes transferred

Direct Connect:
- Reads a specified number of bytes from the printer into the buffer.

Network:
- Calls AppleTalk `PAPRead` to read synchronously. Since there is no completion pointer, reading from a network device must always be done synchronously. To read asynchronously, use `PrDevAsyncRead`.

PrDevWrite $1E13

Description:
Writes the data in the buffer to the printer and calls the completion routine.

Input:	LONG	write completion pointer
	LONG	buffer pointer
	WORD	buffer length

Direct Connect:
- Writes the contents of the buffer to the printer.
- If the completion pointer is `NIL`, then `RTL`. If it is not, then perform a `JSL` to the completion routine.

Network:
- If the completion pointer is `NIL`, then writing will occur synchronously. Otherwise, writing will occur asynchronously.
- Calls AppleTalk `PAPWrite` to transfer the contents of the buffer.
- If the completion pointer is `NIL`, then `RTL` to the caller. Otherwise, perform a `JSL` to the completion routine first, with the error code in the accumulator.

PrDevClose $1F13

Description:
This call is not required to do anything. However, if you allocate any system resources with `PrDevOpen`, you should deallocate them at this time. As with start and shutdown, note that `PrDevClose` could be called without a balancing `PrDevOpen` (the reverse is not true), and you must be prepared for this if you try to deallocate resources which were never allocated.

Input: none

Direct Connect:
- No required function.

Network:
- Sets `End-Of-Write` parameter in AppleTalk `PAPWrite` parameter block to one.
- Calls `PAPWrite` with no data.
- Calls `PAPClose`.

PrDevStatus $2013

Description:

This call performs differently for direct connect and network drivers. For direct connect drivers, it currently has no required function, although it may return the status of the port in the future. For network drivers, it calls an AppleTalk status routine, which returns a status string in the buffer (normally a string like “Status: The print server is spooling your document”).

Input:	LONG	status buffer pointer

Direct Connect:

- Does nothing.

Network:

- Calls AppleTalk `PAPStatus`.

PrDevAsyncRead $2113

Description:

Since `PrDevRead` cannot read asynchronously, this call is provided for that task. Note that this does nothing for direct connect drivers, and if the completion pointer is `NIL`, it behaves for network drivers exactly as `PrDevRead` does.

Input:	WORD	space for result
	LONG	completion pointer
	WORD	buffer length
	LONG	buffer pointer
Output:	WORD	number of bytes transferred

Direct Connect:

- Does nothing.

Network:

- If the completion pointer is `NIL`, then performs exactly as `PrDevRead`.
- Calls AppleTalk `PAPRead`; the actual length read is passed back in the `PAPRead` parameter block.
- Perform a `JSL` to the completion routine, which returns the length read in the `X` register and an EOF flag in the `Y` register. As usual, the accumulator contains the error code and the carry is set if an error occurs.
- In the case of a synchronous call, it performs a `JSL` to the completion routine, which pushes the length read onto the stack.

PrDevWriteBackground $2213

Description:
This routine is not implemented at this time.

Input:	LONG	completion procedure pointer
	WORD	buffer length
	LONG	buffer pointer

PrPortVer $2413

Description:
Returns the version number of the currently installed port driver.

Input:	WORD	space for result
Output:	WORD	Port driver's version number

Direct Connect and Network:
- Gets the internal version number of the port driver and returns it on the stack.

Note:
The internal version number is stored as a major byte and a minor byte (i.e., $0103 represents version 1.3)

PrDevIsItSafe $3013

Description:
This call checks to see if the port or card which your driver controls is enabled. It should check at least the corresponding bit of $E0C02D, and checking the Battery RAM settings wouldn't hurt any either.

Input:	WORD	space for result
Output:	WORD	Boolean indicating if port is enabled

Direct Connect and Network:
- Checks the system to see if the hardware and/or firmware for the card or port this driver controls is enabled, and returns TRUE if it is safe to proceed and FALSE if not. Note that for a port driver that controls an interface card, this call should return FALSE if the card is disabled and the port is enabled, while for a port driver which controls an Apple IIGS internal port, the returned value should be TRUE if the port is enabled and FALSE if not.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1 & 2
- Apple IIGS Technical Note #35, Printer Driver Specifications

Apple IIGS
#37: Free-Form Synthesizer Tips

Revised by: Jim Mensch November 1988
Written by: Jim Mensch May 1988

This Technical Note is intended to help a person who is unfamiliar with the Apple IIGS Sound Tool Set use the Free-Form Synthesizer effectively.

The primary function of the Free-Form Synthesizer is to allow an application program to start one or more complex digitized or computed waveforms playing on the Apple IIGS without further intervention from the application. The waveform is a series of bytes, each representing the amplitude of your outgoing sound at a particular moment in time (defined by the sampling frequency you set). After a call to `FFStartSound`, the Sound Tool Set takes care of all chores involved in loading the DOC RAM, setting up registers, and actually playing your sound. Once playing, your sound will continue until either the Sound Tool Set encounters a `NIL` pointer in the waveform list, or until you call `FFStopSound`.

FFStartSound Parameters

`FFStartSound` has only two parameters: the first a `Word` containing channel, generator, and mode information, and the second a `Pointer` to a parameter block.

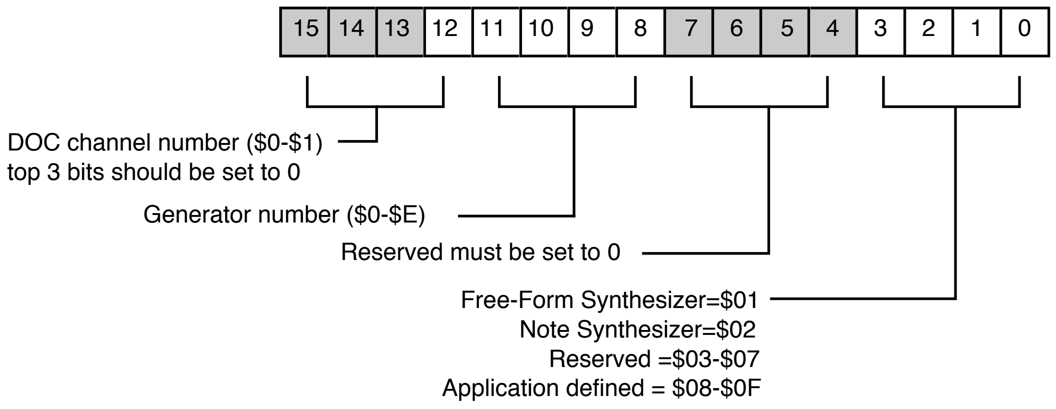

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
DOC channel number ($0-$1)
top 3 bits should be set to 0
Generator number ($0-$E)
Reserved must be set to 0
Free-Form Synthesizer=$01
Note Synthesizer=$02
Reserved =$03-$07
Application defined = $08-$0F

Figure 1 – Channel-Generator-Mode Word

The Channel-Generator-Mode `Word` is broken down into 4 nibbles. The low-order nibble specifies the particular synthesizer you are using. (Because this Note is only about the Free-

Form Synthesizer, we will be using only a 1 in this nibble.) The adjacent nibble must be set to 0 for now. The next nibble specifies which generator to use. The IIGS has 15 generators from which to choose, and as the application designer, it is up to you to decide which one to use. It might be appropriate, however, to call `FFGeneratorStatus` first to ensure that the generator currently is available. (It could be in use already by a desk accessory or previously started sound.) The high-order nibble specifies which channel to use. The IIGS supports two separate sound channels for output. If you are using a stereo adapter, you could start up many sounds and route them to either channel 0 or channel 1 to get a full stereo effect. (The channel is ignored if you are not using a special piece of multi-channel hardware.)

The parameter block contains parameters describing the sound and how it should be played. Here is a sample Pascal definition of that parameter block:

```
FFParmBlock = record
                waveStart:Ptr;
                waveSize:Integer;
                freqOffset:Integer;
                DOCBuffer:Integer;  { High order byte significant }
                bufferSize:Integer; { Low order byte significant }
                nextWave:^FFParmBlock;
                volSetting:Integer;
               end;
```

The first parameter is a 4-byte address telling the Free-Form Synthesizer where in memory it can locate your sample data. The next parameter is a word specifying the number of 256-byte pages of sound you wish to play. The waveform data should be a series of bytes, each representing one sample. Wave tables must be exact multiples of 256 bytes.

Note: A zero value in the waveform can cause a sound to stop, so be sure to check your data to ensure that this does not happen.

The frequency offset parameter specifies the sampling frequency that the Free-Form Synthesizer should use during playback. This number can be computed by the following formula:

`freqOffset` = ((32*Sample rate in Hertz)/1645)

The frequency offset parameter is the most often misunderstood parameter, so I will explain a little about sampling rates. The sampling rate is how many samples (bytes) per second to play. If you have a digitized wave that represents 2 seconds of sound, and it takes up 44K of memory, then it was sampled at 22 kHz (which, by the way, is good for full sound reproduction). The sampling rate must be at least twice that of the maximum fundamental frequency you want to sample. However, for good sound reproduction, you may want to sample at least eight times the fundamental frequency in order to capture the higher harmonics of musical instruments and the human voice.

The DOC starting address and buffer size tell the Free-Form Synthesizer which portion of the 64K sound RAM to use as a buffer during playback. The wave is taken from your waveform in chunks and placed in sound RAM for playback. Each time the buffer nears empty, it will need to be reloaded with more sound. The size of the buffer specified determines how often the Free-Form Synthesizer must interrupt the 65816 to reload the buffer. The buffer size must be a power of two because of the way the sound General Logic Unit (GLU) specifies addresses. (The value for this parameter must also be a power of two.) A good length to use would be at least 1/10 second of sound. For example, if you were using a sampling rate of 16 kHz (16,000 samples per second), you would want a buffer at least 2,048 bytes long, or about 8 pages. It does not hurt to round this number up. You manage the DOC RAM, so you should decide what memory to use. It is usually a good idea to have multiple buffers if you have a chain of waves. (I like leaving page zero free, as the Note Synthesizer uses the data in the first 256 bytes, and accidentally placing a zero in that page could cause it to fail.)

The next wave pointer is a 4-byte pointer to the next parameter block. With this parameter you can string together many waveforms for more continuous sound, or you can make your sounds infinitely recursive by pointing back to the original wave form.

The volume setting is a word which represents the relative playback volume. It can range from 0 to 255.

Other Tips

When you shut down the Sound Tool Set, it will stop all pending sounds, so be sure to leave ample time between starting and ending a sound. If you have a series of wave forms strung together, you can change their parameters on the fly. Changes take effect as soon as the waveform is started. (You could use this to find the correct sampling frequency of a wave, by having the next wave pointer point back to the start of your parameter block. This would cause the sound to play indefinitely. You then could change the `freqOffset` value, and the sound would change each time it is restarted.)

Here is a sample code segment (in APW Assembler format) that creates a 1-kHz wave in memory sampled at 16 kHz and plays it:

```
FFSound         DATA

theSound        ds      $2000                 ; FFSound wave...
MyFFRecord      dc      A4'theSound'          ; address of wave
                dc      i'$20'                ; size of wave in pages..
Rate            dc      i'311'                ; 16-kHz sample rate
                dc      i'1'                  ; DOC starting address
                dc      i'$0800'              ; DOC buffer size
                dc      a4'0'                 ; no next wave
Vol1            dc      i'$007F'              ; kinda medium..

; 1-kHz triangle wave sampled at 16 kHz one full segment
oneAngle        dc      i1'$40,$50,$60,$70,$80,$90,$A0,$B0'
                dc      i1'$C0,$B0,$A0,$90,$80,$70,$60,$50'
                End

TestFF          Start
                Using   FFSound
MakeWave        ANop
                ldx     #$0000
MW0010          txa                           ; get index
                and     #$000F                ; use just low nibble as index
                tay                           ; into triangle wave table
                lda     oneAngle,y            ;
```

```
            sta    theSound,X          ; and store it into sound buf
            inx
            inx
            cpx    #$2000              ; we Done?
            blt    MW0010              ; nope better finish
            PushWord     #$0001
            PushLong     #MyFFRecord
            _FFStartSound
            rts
            end
```

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2

Apple IIGS
#38: List Controls in Dialog Boxes

Revised by: C.K. Haun — September 1990
Written by: Keith Rollin, Dave Lyons & Eric Soldan — May 1988

This Technical Note describes how to include a list control into a dialog box. Sample APW C source code is included.
Changes since March 1990: Changed input parameter definition for `myFilterProc` from long pointer to word pointer.

The need to put a list control into a dialog box is obvious. The Print Manager does it. The Font Manager does it. You may want to use one in your own application to manage a list of data base fields or spreadsheet functions. However, performing the task is not as obvious as the need.

Given the features of TaskMaster in System Software 5.0, it is now much easier to emulate a modal dialog in a normal window. If you need to add a list control to a modal dialog, you should seriously consider emulating a modal dialog with a normal window instead of using the Dialog Manager. If you use the Dialog Manager, the following procedure and sample C fragment illustrate the technique necessary for adding a list control.

Note that only **one** list control is allowed in a modal dialog. If you need more than one, the Dialog Manager cannot help you—create a normal window instead.

Individual Steps

Basically, there are three check-off items for putting a list control into a dialog box:

1. You must install the list explicitly into the dialog box yourself. This should be done **after** you have created the dialog box with a call to `NewModalDialog` or `GetNewModalDialog`. Do **not** install it as a `UserItem` or `UserCtlItem`. Installing it as a `UserItem` would cause the Dialog Manager to place an invisible custom control over the list, preventing later use of `FindControl` to manage it. Installing the list as a `UserCtlItem` does not allow the list control to be properly initialized.

   Note: After you add the list control, you must **not** add any more dialog items.

   ```
   InitValues()
   ```

```
        {
            /* Get a Full Screen, invisible dialog window with only
               a Quit button in it*/
            myDialog = GetNewModalDialog(&PrintDialog);
        /* Add this List Control ourselves */
        myListHndl = CreateList(myDialog,&myList);

        /* Get the handle for the Scrollbar Control */
        listScrollHandle = (**myListHndl).ctlListBar;

        /* Save and Zero out the RefCons */
        listRefCons = GetCtlRefCon(myListHndl);
        scrollRefCons = GetCtlRefCon(listScrollHandle);
        ZeroRefCons(); /* This is explained below in item #3 */

        /* Now show the dialog box */
        ShowWindow(myDialog);
    }
```

2. Because the list control is not a dialog item, a custom `FilterProc` must be installed for `ModalDialog` to test for mouse-down events. Pass the address of this routine (with the high bit set so that default handling of items is in effect) when you call `ModalDialog`.

```
pascal Word myFilterProc(theDialog, theEvent, theItem)
    GrafPortPtr    theDialog;
    EventRecord    *theEvent;
    word           *theItem;

{
    CtlRecHndl  tHandle;

    if ((*theEvent).what == mouseDownEvt) {
        FindControl(&tHandle,(*theEvent).where,theDialog);
        if ((tHandle == myListHndl) || (tHandle == listScrollHandle)) {

            /* Set the RefCons back to the way the list manager likes them */
            RestoreRefCons();
            TrackControl((*theEvent).where,(LongProcPtr) -1, tHandle);
            ZeroRefCons();

            /* Tell the Dialog Manager that we handled this event */
            return(true);
        }
    }
    /* We didn't do anything, so return false to get Dialog Manager
        to handle this event */
  return(false);
}
```

3. The Dialog Manager uses the `RefCon` field of its items (all of which are installed as controls). Unfortunately, the List Manager also uses the `RefCon` field for its own purposes. This shared use means that a judicious juggling of those values is required. This juggling is the reason for the two routines `RestoreRefCons` and `ZeroRefCons` used above.

```
/* Zero out the RefCons for the Dialog Manager */
ZeroRefCons()
{
    SetCtlRefCon(0,myListHndl);
    SetCtlRefCon(0,listScrollHandle);
}
```

```
       /* Restore the RefCons for the List Manager */
       RestoreRefCons()
       {
           SetCtlRefCon(listRefCons,myListHndl);
           SetCtlRefCon(scrollRefCons,listScrollHandle);
       }
```

Note: Because the Dialog Manager currently uses the `RefCon` to keep track of which dialog item is identified with which particular control, zeroing the `RefCon` fields can cause a little confusion. Specifically, those who would like to do `GetFirstDItem` from within a Standard File call may get a zeroed `RefCon` as a result. This is true for Standard File 3.0 and later (System Software 5.0), as this is the first implementation of Standard File to use the List Manager.

Putting It All Together

Here are most of the pieces put together. `InitTools` and `ShutDownStuff` routines have been omitted, but they are straightforward.

```
char              **y,*z;
GrafPortPtr       myDialog;
ListCtlRecHndl    myListHndl;
CtlRecHndl        listScrollHandle;
long              listRefCons, scrollRefCons;

#define Quit      ok

char  quitStr[] = "\pQuit";

ItemTemplate quitButton =  {
           Quit,
           140,450,154,590,
           buttonItem,
           quitStr,
           0,
           0,
           NULL};

DialogTemplate PrintDialog = {
           30,20,190,620,
           false,
           0,
           &quitButton,
           NULL};

char string1[] = "String1";
char string2[] = "String2";
char string3[] = "String3";
char string4[] = "String4";
char string5[] = "String5";
char string6[] = "String6";
char string7[] = "String7";
char string8[] = "String8";
```

```
MemRec  myMembers[8] = {
           string1, 00,
           string2, 00,
           string3, 00,
           string4, 00,
           string5, 00,
           string6, 00,
           string7, 00,
           string8, 00};

ListRec  myList = {
           40,175,102,400, /* Enclosing Rectangle */
           8,              /* Number of List Members */
           6,              /* Max Viewable members */
           3,              /* Bit Flag */
           1,              /* First member in view */
           NULL,           /* List control's handle */
           NULL,           /* Address of Custom drawing routine */
           10,             /* Height of list members */
           5,              /* Size of Member Records */
           (MemRecPtr)myMembers,/* Pointer to first element in MemRec[] */
           NULL,           /* Becomes Control's refCon */
           NULL            /* Color table for list's scroll bar */
           };

/* ************************** */

main()
{
       word what;

       InitTools();         /* initialize tools */
       InitValues();        /* Get dialog box. Install List control */
       do {
           what = ModalDialog((WordProcPtr)((long)myFilterProc | 0x80000000));
       } while (what != Quit);
       ShutDownStuff();
}

pascal Word myFilterProc(theDialog, theEvent, theItem)
       GrafPortPtr    theDialog;
       EventRecord    *theEvent;
       word           *theItem;

{
       CtlRecHndl     tHandle;

       if ((*theEvent).what == mouseDownEvt) {
           FindControl(&tHandle,(*theEvent).where,theDialog);
           if ((tHandle == myListHndl) || (tHandle == listScrollHandle)) {

               /* Set the RefCons back to the way the list manager likes them */
               RestoreRefCons();
               TrackControl((*theEvent).where,(LongProcPtr) -1, tHandle);
               ZeroRefCons();

               /* Tell the Dialog Manager that we handled this event */
               return(true);
           }
       }
       /* We didn't do anything, so return false to get Dialog Manager
          to handle this event */
  return(false);
}
```

```
/* Zero out the Refcons for the Dialog Manager */
ZeroRefCons()
{
       SetCtlRefCon(0,myListHndl);
       SetCtlRefCon(0,listScrollHandle);
}

/* Restore the Refcons for the List Manager */
RestoreRefCons()
{
       SetCtlRefCon(listRefCons,myListHndl);
       SetCtlRefCon(scrollRefCons,listScrollHandle);
}

InitValues()
{
       /* Get a Full Screen, invisible dialog window with only a Quit button in it*/
       myDialog = GetNewModalDialog(&PrintDialog);

       /* Add this List Control ourselves */
       myListHndl = CreateList(myDialog,&myList);

       /* Get the handle for the Scrollbar Control */
       listScrollHandle = (**myListHndl).ctlListBar;

       /* Save and Zero out the RefCons */
       listRefCons = GetCtlRefCon(myListHndl);
       scrollRefCons = GetCtlRefCon(listScrollHandle);
       ZeroRefCons();

       /* Now show the dialog box */
       ShowWindow(myDialog);
}
```

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#39: Mega II Video Counters

Revised by: Dave Lyons July 1989
Written by: J. Rickard May 1988

This Technical Note describes the Mega II video output registers, which your applications can use to get information about where the beam is located on the Apple IIGS display.
Changes since November 1988: Corrected description of when VBL begins and simplified example code to read the scan line number.

The Mega II controls video timing for the Apple IIGS with a 16-bit counter split into a 7-bit horizontal and a 9-bit vertical part (Figure 1). The counter outputs are made available to programs running on the machine through two addresses in the I/O space, $C02E for the vertical count and $C02F for the horizontal count. These outputs can be used by a program for finer control over display update timing.

Vertical Counter									Horizontal Counter						
V5	V4	V3	V2	V1	V0	VC	VB	VA	HPE	H5	H4	H3	H2	H1	H0
$E0C02E								$E0C02F							

Figure 1 – Mega II Video Counter

You can see that one bit of the nine-bit vertical counter is in location $E0C02F with the seven bits of the horizontal counter. Keep this location in mind when reading the counters.

The seven-bit horizontal counter starts at $00 and counts from $40 to $7F (the sequence is $00, $40, $41,…,$7E, $7F, $00, $40,…). The active video time consists of 40 one μsec clock cycles starting with $58 and ending with $7F. Since this count changes at 980 nanosecond intervals, it will probably be of little use to most programs.

The nine-bit vertical counter ranges from $FA through $1FF (250 through 511) in NTSC mode (vertical line count of 262) and from $C8 through $1FF (200 through 511) in PAL video timing mode (vertical line count of 312). Vertical counter value $100 corresponds to scan line zero in NTSC mode. The vertical count changes at 63.7 μsec intervals, giving a program time to respond to a specific count before it changes. The vertical counter **byte**, at $E0C02E, only changes half as often (at 127 µsec intervals) since the lowest bit of the nine-bit counter is actually stored in the next byte (at $E0C02F).

The nine-bit counter consists of bits VA, VB, VC, V0, V1, V2, V3, V4 and V5. Bits V0 through V5 can be read as a six-bit value. If this value is between 0 and 23, it is the line on the text screen currently being updated. Other values indicate the vertical blanking cycle is occurring. Bits VA through VC can be read as a three-bit value (0-7) indicating which scan line of a text character (characters are composed of eight lines) is currently being drawn.

The vertical counter can also be used to determine which scan line (0-191 for most video modes, including high-resolution and double high-resolution, and 0–199 for super high-resolution) is being updated at any given moment.

Example

Suppose you want to repaint a portion of the super high-resolution screen that will require more time than the vertical blanking period allows. You will have a tear in your animation when the screen's refresh cycle catches up with your drawing.

One solution to this problem would be locating the approximate place the tear occurs and starting your drawing when the system is scanning that line of graphics. Let's say you are painting an area that is about (for example) 100 pixels wide and 200 pixels tall in 320 mode, and that the tear will occur somewhere around scan line 80. To avoid the tear, you would wait until the system is scanning line 80, then you would start redrawing at the top of the screen. This way, you should be finished drawing when the system is back to scanning line 80 again and you will have flicker-free screen updating.

The tricky part is trying to determine just when the system is scanning any given scan line. One way to determine this is to examine the Mega II video counter registers at $E0C02E (vertical) and $E0C02F (horizontal), described above. By using some simple arithmetic you can come up with the exact scan line being updated. The following piece of code computes the current scan line number (assuming eight-bit native mode):

```
lda    >$E0C02F
asl    A                  ;VA is now in the Carry flag
lda    >$E0C02E
rol    A                  ;roll Carry into bit 0
```

The result (in `A`) is the low byte of the vertical counter. This value is 0 for the first scan line, 1 for the second scan line, etc. Values $FA to $FF are used twice, since you ignore the high byte of the vertical counter. (The six scan lines immediately above scan line 0 are numbered $0FA to $0FF, and the six above those are $1FA to $1FF.) The example code leaves the highest bit of the vertical counter in the Carry flag, if you really want it.

Note that the VBL interrupts always trigger at scan line 192, even in Super Hi-Res display mode, and that the $C019 soft switch indicates vertical blanking is in effect starting at scan line 192. Be careful polling for a specific scan line number—if interrupts are enabled, it is conceivable that the system will be busy processing an interrupt every time that scan line is being scanned, so your program will hang forever waiting for it.

Setting a scan line interrupt is another way to determine when a particular super high-resolution scan line is being drawn. However, you must be careful in turning scan line interrupts on and off so that you do not interfere with the cursor in QuickDraw II (which uses scan line interrupts).

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2
- Apple IIGS Technical Note #40, VBL Signal

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#40: VBL Signal

Revised by: Dave Lyons July 1989
Written by: Rob Moore & Rilla Reynolds May 1988

This Technical Note discusses reading the VBL signal to accomplish smooth animation.
Changes since November 1988: Noted that vertical blanking does not begin when you might expect on the Apple IIGS and removed references to the Apple IIc.

Applications can accomplish smooth animation on the Apple IIGS and Apple IIe by changing the data on the screen during the time the system is tracing the unusable area of the display. This time is called "vertical blanking" or "VBL" in this Note. You can determine the state of the VBL signal by reading location $C019.

On the Apple IIGS, the $C019 sense of the VBL signal differs from the IIe. On the IIGS, the screen is blanked when the most significant bit of $C019 is **high** (greater than 127 or $7F), while on the IIe, the screen is blanked when the bit is **low** (less than 128 or $80).

A VBL interrupt also is available on Apple II systems via the Apple IIGS Miscellaneous Tool Set or mouse firmware, the Apple IIe mouse card, and the Apple IIc mouse firmware.

On the Apple IIGS, vertical blanking begins at scan line 192 regardless of the display mode. When the Super Hi-Res display is visible, vertical blanking begins eight scan lines before the bottom of the display area. If the VBL interrupt is enabled, it triggers at scan line 192.

Further Reference

- Apple IIGS Technical Note #39, Mega II Video Counters

Developer Technical Support

Apple IIGS
#41: Font Family Numbers

Revised by: Matt Deatherage & Keith Rollin November 1990
Written by: Rilla Reynolds & Jeff Erickson May 1988

This Technical Note lists fonts and font family numbers as well as considerations when printing to a LaserWriter printer and a word of caution about using font family numbers.
Changes since November 1988: Added information about the font family numbering convention used by those who assign font family numbers.

The following table lists fonts and their corresponding font family numbers. All family numbers are listed in decimal format except the first three.

ID	Family Name	ID	Family Name
$FFFD	Chicago	12	Los Angeles
$FFFE	Shaston	13	Zapf Dingbats®†
$FFFF	(no font)	14	Bookman®†
0	System Font	15	Helvetica Narrow†
1	System Font	16	Palatino®†
2	New York	18	Zapf Chancery®†
3	Geneva	20	Times®†
4	Monaco	21	Helvetica®†
5	Venice	22	Courier†
6	London	23	Symbol†
7	Athens	24	Taliesin
8	San Francisco	33	Avant Garde®†
9	Toronto	34	New Century Schoolbook†
11	Cairo		

Fonts denoted with a cross (†) are resident in the ROM on the LaserWriter Plus, IINT and IINTX printers. The name of Times on these printers is actually Times-Roman. The decimal font family ID for Shaston (a modified Helvetica) is 65534 (–2), not 65524 as documented in the Font Manager chapter of the *Apple IIGS Toolbox Reference*.

When printing to a LaserWriter printer with the font substitution option turned on, the system substitutes Times, Helvetica, and Courier for the screen fonts New York, Geneva, and Monaco respectively.

Prior to System Software 3.2, all non-LaserWriter fonts (except New York, Geneva, and Shaston) were converted to Courier when printing. With System Software 3.2 and later, the LaserWriter driver prints bitmap versions of the screen fonts if they are non-LaserWriter fonts unless it is driving an original LaserWriter printer. In this case, fonts which are in ROM on later LaserWriter printers are converted to Courier unless you download a PostScript® version of the font prior to printing. This difference is a limitation of the current LaserWriter driver and it occurs even if the font substitution option is turned off. With System Software 5.0 and later, the LaserWriter driver uses fonts previously downloaded, although it does not download PostScript fonts itself.

Font Family Number Conventions

By convention, font family numbers that have the high bit set are designed for the 5:12 aspect ratio of the Apple IIGS computer. Font family numbers with the high bit clear are designed for computers with a 1:1 pixel aspect ratio, such as the Macintosh. Fonts designed for a 1:1 pixel aspect ratio appear “tall and skinny” when displayed on an Apple IIGS.

Some third-party font packages were released before this convention was defined; therefore, font family numbers between 1000 and 1200 (decimal) do not adhere to this convention.

Caution

Font family numbers can be arbitrary numbers which the system assigns to fonts. We recommend that you always ask for a font by name (with the Font Manager call `GetFamNum`), then use the returned family number as input to those calls which require it. (On the Macintosh, the Font/DA Mover checks to see if a font family number is already in use by the system when it installs fonts. If it finds that a number is already in use, it changes the current font number to an unused number. If you move a font from the Macintosh to the IIGS, the font family number is likely to be arbitrary, as is the font family number of any user-created fonts.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1-3
- Apple IIGS Technical Note #67, LaserWriter Font Mapping

Apple IIGS
#42: Custom Windows

Written by: Dan Oliver & Keith Rollin

November 1988

This Technical Note describes custom windows which are now supported with Window Manager version 2.2. This Note supersedes all prior documentation on custom windows.

With Window Manager version 2.2 or later, which is available on Apple IIGS System Disk 3.2 and later, you may now define your own type of window or window shape, such as a round or hexagonal window. You also may define a window which performs tasks that would normally be handled by an application.

To define your own type of window, a custom window, you must write a routine that performs some window functions. This routine is a window definition procedure (`defProc`), and in this case it is a custom window `defProc`. When the Window Manager needs to do something window specific, it calls your `defProc`.

The window `defProc` is a good part of the Window Manager, and writing one is not an easy task. A window `defProc` must perform complicated tasks that are very dependent on the state of the machine, and it must be very careful not to disturb the state of the machine. One of the problems in writing a `defProc` is knowing when it can do something and when it cannot. It is almost impossible to document all of the combinations of calls that you can or cannot make from one part or another of the `defProc`, and even if all cases were found, the resulting document would read like something from an obscure government bureau and probably be even harder to understand.

Now that you know writing a `defProc` is tough, here's how to make things as easy as possible. Try to understand how the system interacts with the `defProc` and work with the system. For example, a `defProc` is called to hit test window parts when the user presses the mouse button. The Window Manager will pass that part back to the `defProc` to perform drawing while the Window Manager is tracking the pressed button. The `defProc` could keep control when asked to hit test and perform the tracking itself, but since this is not how the system is designed to work, your `defProc` will be hard to write, may not ever work correctly, and may break in future versions of the Window Manager. Try to stay on the path outlined in this Technical Note. Also understand that the interface to definition procedures is as general as possible to allow them to perform tasks which are as yet unknown. To allow for this future growth, the outlined path is not always a clear path.

Another way to make things easier is to write conservative code. Do not assume things like the data bank being set to something nice when the `defProc` is called or the caller restoring the direct page pointer upon return if you have changed it. Use caution. A `defProc` can be very difficult to debug because it is not very linear and can be called when you least expect.

Interaction Between the Window Manager and TaskMaster

The Window Manager and `TaskMaster` actually do much less than many people think since window definition procedures perform most of the tasks. The definition procedures handle such things as title bars, information bars, and scroll bars, while the Window Manager and `TaskMaster` support these things by passing requests to the `defProc` in standard ways. The Window Manager knows that windows have some shape, overlap, may contain parts, may be invisible, and are created and deleted, but it does not know much else. `TaskMaster` knows to call `GetNextEvent` and performs some tasks, but much of what many people consider `TaskMaster` is contained in the standard document window `defProc`. In addition to the list mentioned above, the `defProc` handles calling `TrackGoAway` and scrolling the content. The remainder of this Note describes what is expected of a `defProc` and when.

Telling the Window Manager About Your Window

You tell the Window Manager about your custom window when `NewWindow` creates it. Instead of passing the parameter list defined in `NewWindow`, you pass a pointer to a custom window parameter list. A custom window parameter list is defined as follows:

`paramID`	WORD	ID of parameter list, zero for custom.
`newDefProc`	LONG	Address of your custom defProc.
`newData`	BYTE[n]	Additional data defined by your defProc.

`NewWindow` checks the `paramID` field and calls your `defProc` with the pointer to the parameter list. See the `wNew` operation under Calling the Custom DefProc for more information.

Once `NewWindow` creates the window, the Window Manager will always know that it is defined by your `defProc`.

Calling the Custom defProc

A window `defProc` is called with the following items on the stack:

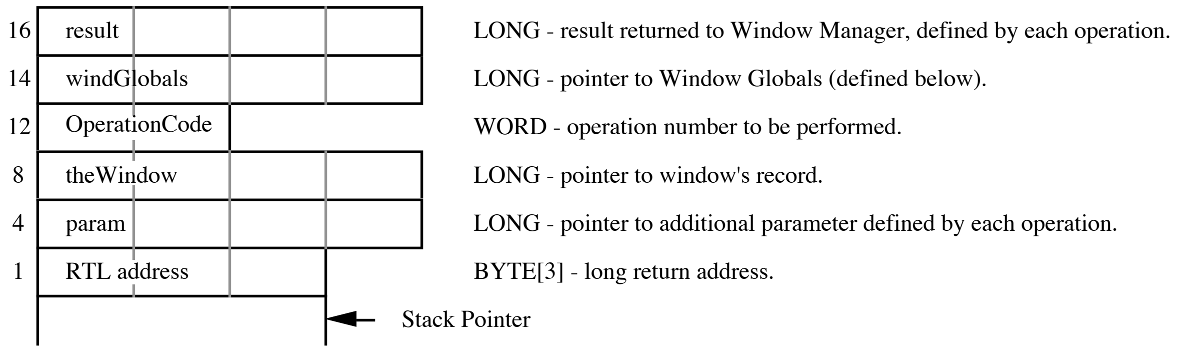

16
result
14
windGlobals
12
OperationCode
8
theWindow
4
param
1
RTL address
Stack Pointer
LONG - result returned to Window Manager, defined by each operation.
LONG - pointer to Window Globals (defined below).
WORD - operation number to be performed.
LONG - pointer to window's record.
LONG - pointer to additional parameter defined by each operation.
BYTE[3] - long return address.

Figure 1 – Stack Prior to Calling a Window defProc

The defProc must return with the carry flag clear if there was no error or with the carry flag set and the y register set with an error code if there was an error.

Window globals (`windGlobals`) is a pointer to a table of variables which the Window Manager maintains for use by the `defProc`. The table is defined as follows:

`lineW`	WORD	Width of vertical lines (size depends on video mode).
`titleHeight`	WORD	Height of a standard title bar.
`titleYPos`	WORD	Y offset for the title (in system font) to center in a standard title bar.
`closeHeight`	WORD	Height of the close box icon.
`closeWidth`	WORD	Width of the close box icon.
`defWindClr`	LONG	Pointer to the default window color table.
`windIconFont`	LONG	Handle of the current window icon font.
`screenMode`	WORD	TRUE if 640 mode, FALSE if 320 mode.
`pattern`	BYTE[32]	Temporary pattern buffer.
`callerDpage`	WORD	Direct page pointer of the last caller to `TaskMaster`.
`callerDataB`	WORD	Data bank of the last caller to `TaskMaster` (bank in both bytes).

Operation numbers are as follows (each operation is described later in its own section):

`wDraw`	0	Draw the window's frame.
`wHit`	1	Tell in what region the mouse button was pressed.
`wCalcRgns`	2	Calculate `wStrucRgn` and `wContRgn`.
`wNew`	3	Complete the creation of a window.
`wDispose`	4	Complete the disposal of a window.
`wGetDrag`	5	Return address that will draw the outline of the window while dragging.
`wGrowFrame`	6	Draw the outline of a window being resized.
`wRecSize`	7	Return size of the additional space needed in the window record.
`wPosition`	8	Return `RECT` that is the window's `portRect`.
`wBehind`	9	Return where the window should be placed in the window list.
`wCallDefProc`	10	Generic call to a `defProc`, defined by the `defProc`.

wDraw, Operation 0

The `wDraw` operation draws the window's frame and is only called for visible windows. This operation draws in local coordinates in the current `GrafPort`, which is the Window Manager's `GrafPort`. When the drawing is finished, the only states of the `GrafPort` that may have changed are the pen pattern, the fill pattern, and the pen size, as all other states must be the same as when the `defProc` was called. This means that if you change the font to print some text, you must save and restore the original font. For the pen, `PenNormal` will restore the pen to an acceptable state.

`Param` is defined as follows:

Bit 31	1 to highlight the indicated part, 0 to unhighlight.	
Bits 0-30	The part to draw (either highlighted or unhighlighted):	
	0	Draw the window's entire frame, including any frame controls and the items listed below. Note that you should check the window's `fHilited` flag to determine how to draw the frame.
	1	Draw the go-away region.
	2	Draw the zoom region.
	3	Draw the information bar.

`Result` returned must be zero and the carry flag must be clear.

The Window Manager will draw the content.

Need to Redraw Your Window?

If your custom window `defProc` gets called to change some item in its window record (see `wCallDefProc` below), you may want to redraw your window. For instance, if your application makes a `SetWTitle` call, you would want to draw the name of the new title on the screen.

The routine `wCallDefProc` can call the `wDraw` routine to do this drawing. However, it should bracket the calls to `wDraw` with two Window Manager calls that save and restore some internal variables:

`StartFrameDrawing`	$5A0E
`PUSH:LONG`	Pointer to the window record (not the `GrafPort`)

This call does the setup for drawing a window frame and is only called by a window definition procedure before drawing the frame. You should call `EndFrameDrawing` when finished drawing.

`EndFrameDrawing`	$5B0E
No input or output	

This call restores the Window Manager variables after a call to StartFrameDrawing and is only called by a window definition procedure after drawing a window frame.

wHit, Operation 1

The `wHit` operation is called to hit test the window's frame. Given a set of screen coordinates, this operation should return what part, if any, of the window is at that coordinate. This operation is only called for visible windows. The current port will be that of the Window Manager and the window frame will be in local coordinates.

`Param` is defined as:

Bits 0-15	Vertical (Y) coordinate in local coordinates.
Bits 16-31	Horizontal (X) coordinate in local coordinates.

`Result` returned must be one of the following values and the carry flag must be clear:

`wNoHit`	0	Not on the window at all.
`wInDrag`	20	Coordinates are in the window's drag region (title bar).
`wInGrow`	21	Coordinates are in the window's grow region (size box).
`wInGoAway`	22	Coordinates are in the window's go-away region (close box).
`wInZoom`	23	Coordinates are in the window's zoom region (zoom box).
`wInInfo`	24	Coordinates are in the window's information bar.
`wInFrame`	27	Coordinates are in the window, but not in any of the other areas.

xx Any code the application can handle (bit 15 is reserved for theWindow Manager)

wCalcRgns, Operation 2

The wCalcRgns operation, which is called only for visible windows, is used to calculate the window's entire region (frame plus content called StrucRgn) and just its content region (called ContRgn). Both regions must be set to global coordinates, and both will already be allocated with their handles stored in the window record's wStrucRgn and wContRgn fields.

Use the portRect and the boundsRect of the window's GrafPort to calculate these two regions. The port will have been set from the information passed to NewWindow along with any size changes. A method for obtaining the global RECT of the content is given below. Refer to the QuickDraw II chapter in the *Apple IIGS Toolbox Reference* for a full description of ports. When calculating the regions, do not change the clip region (ClipRgn) or the visible region (VisRgn) of the GrafPort.

Param is not defined and should not be used.

Result returned must be zero and the carry flag must be clear.

```
IN:     window = pointer to window record.
OUT:    rect = global RECT of window's content.

        ldy #wPort+portRect+y1
        lda [<window],y
        ldy #wPort+portInfo+boundsRect+y1
        sec
        sbc [<window],y
        sta <rect+y1
;
        ldy #wPort+portRect+x1
        lda [<window],y
        ldy #wPort+portInfo+boundsRect+x1
        sec
        sbc [<window],y
        sta <rect+x1
;
        ldy #wPort+portRect+y2
        lda [<window],y
        ldy #wPort+portInfo+boundsRect+y1
        sec
        sbc [<window],y
        sta <rect+y2
;
        ldy #wPort+portRect+x2
        lda [<window],y
        ldy #wPort+portInfo+boundsRect+x1
        sec
        sbc [<window],y
        sta <rect+x2
```

Although there are other ways to obtain the global RECT of the content, this example gives the correct method. You should never rely on the top and left side of the portRect being zero.

wNew, Operation 3

The `wNew` operation is called to perform any additional initialization that may be required for a custom window. The following items are already done for the window:

- If a window record is supposed to be allocated, it is. All fields, other than those fields listed below, are set to zero
- A port opens in the window record's `wPort` field.
- The window is added to the Window Manager's window list, and the `wNext` field is set.
- The `wDefProc`, `wStrucRgn`, `wContRgn` and `wUpdate` regions are set with the handles of the allocated regions. It is the responsibility of the `defProc` to define the shape of the `wStrucRgn` and `wContRgn` regions.
- The `fAllocated` and `fHilited` bits in the `wFrame` field of the window record are set (see the window record definition for a definition of these bits) and should not be disturbed; all other bits in `wFrame` are set to zero. The `defProc` should set the `fCtlTie`, `fVis` and `fQContent` bits, and it can set and use other bits in the `wFrame` field as it wishes.
- It is the responsibility of the `defProc` to set the `wRefCon`, `wContDraw`, and `wFrameCtls` fields, the bits already mentioned in the `wFrame` field, and any other fields which it defines in the `wCustom` part of the window record.

`Param` is a pointer to the parameter list pointer which was passed to `NewWindow`.

`Result` returned must be zero and the carry flag must be clear.

wDispose, Operation 4

The `wDispose` operation is called to perform any additional disposal that may be required of a custom window. This operation is called before the Window Manager performs any disposal actions on the window.

`Param` is not defined and should not be used.

`Result` should be FALSE to continue disposal or TRUE to abort the disposal. In either case, the carry flag should be clear. Returning TRUE would be very unusual and should be carefully thought out. After returning FALSE, the Window Manager will erase the window, remove the window from the Window Manager's window list, free any controls in the window's `wControls` and `wFrameCtl` lists, free the handles in the `wStrucRgn`, `wContRgn` and `wUpdateRgn` fields, close the window's `GrafPort`, and free its record if it is allocated (see the `wFrame` field).

wGetDrag, Operation 5

The `wGetDrag` operation is called to get the address of a routine that will draw an outline of the window.

`Param` is not defined and should not be used.

`Result` returned must be the address of a frame outline routine or zero for a default frame; the default frame is the bounds `RECT` of the `strucRgn`. The frame outline routine is called from `DragRect` with `dragRectPtr` set to the bounds `RECT` of the `strucRgn`. Your routine is called with the following parameters:

```
PUSH:WORD - delta X
PUSH:WORD - delta Y
PUSH:BYTE[3] - return address
```

Your routine should draw or erase the outline of the object in its new position using the passed deltas. You have several different methods of determining whether to erase or draw and how to compute the position of the object, the easiest method being to draw the outline using `XOR` mode. The first time your routine is called, you draw. The next time your routine is called, you erase. Your routine should draw in the current port. The current pen pattern will be the pattern pointed to by `dragPatternPtr` from `DragRect` and the pen mode is `XOR`.

You also need to know where to draw the outline. One way is to offset the starting `RECT` (`dragRectPtr`) by the given deltas. You should make a copy of the bounds `RECT` of the `strucRgn` when `wGetDrag` is called. Modify that rectangle with the deltas to obtain the rectangle to frame.

wGrowFrame, Operation 6

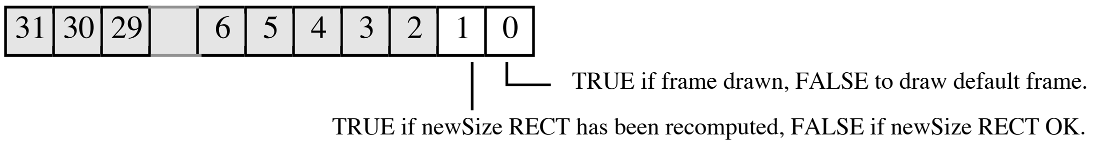
The `wGrowFrame` operation is called to draw an outline of the window when the window is being resized.

This operation should use the current port, pen pattern, and pen mode. The frame should be drawn with only the following QuickDraw II calls: `Line`, `LineTo`, `FrameRect`, `FrameRgn`, `FramePoly`, `FrameOval`, `FrameRRect`, and `FrameArc` (the Invert equivalents to Frame could also be used). You want to use the current `GrafPort` setting with only certain QuickDraw II calls since this routine will be called an even number of times; the first time it is called to draw the frame and the next time to erase that which it drew the first time. If it needs to use QuickDraw II calls other than those listed above, this operation handler could keep track of odd and even calls to know whether to draw or erase the frame.

`Param` is a pointer to the following parameter list:

`newSize`	RECT	Rectangle that defines the new size.
`drawFlag`	WORD	TRUE to draw the frame, FALSE to erase.
`startRect`	RECT	Bounds of `wStrucRgn` when dragging started.
`deltaY`	WORD	Vertical movement since starting to drag (signed).
`deltaX`	WORD	Horizontal movement since starting to drag (signed).

`Result` should be:


31
30
29
6
5
4
3
2
1
0
TRUE if frame drawn, FALSE to draw default frame.
TRUE if newSize RECT has been recomputed, FALSE if newSize RECT OK.

The Window Manager assumes that the frame of the grow outline is the same as the bounds of the window's `wStrucRgn`. This `RECT` is stored in the `startRect` of the parameter list and does not change through out the dragging. The next assumption is that the window grows from the lower right corner. As the cursor moves, the lower right corner of the `RECT` in `newSize` changes. However, if these assumptions are not correct for a custom window they can be overridden by changing the `RECT` in `newSize` (by using `startRect` or the window's record and the deltas) and returning TRUE for bit 1 in `Result`. The carry flag should return clear.

wRecSize, Operation 7

The `wRecSize` operation is called to ask how large a window record should be allocated.

Note: The window pointer passed in `theWindow` is not valid for this call.

`Param` is the parameter list pointer that is passed to `NewWindow`.

`Result` is the number of additional bytes required in the window record. The standard window record header will always be allocated.

Example:

If your custom window needs a one word field in the window record for your own use you would return 2 in `Result`. The Window Manager takes `Result` and adds to it the size of the standard record header of 212 bytes and allocates a window record that is 214 bytes long in this case. Your one word field is at the end of the standard window record header with an offset of 212 bytes.

If there is some error, return the carry flag set with an error code in the y register, which will cause `NewWindow` to abort and return the error code to the application which called it. If there is no error, return the carry flag clear.

Window Record Already Allocated?

If the window record is already allocated then `Result` should be the pointer to the window record with bit 31 of the pointer set to TRUE. Generally, window records are allocated (refer to Window Record Definition at the end of this Note for more information about window records).

wPosition, Operation 8

`Param` is the parameter list pointer that is passed to `NewWindow`.

`Result` is a pointer to the `RECT` that will be the window's `portRect`, and you should return the carry flag clear.

wBehind, Operation 9

`Param` is the parameter list pointer that is passed to `NewWindow`.

`Result` is where the window should be placed in the window list. A long $FFFFFFFF means insert the window as the top window while a long $00000000 means to insert it as the bottom window. Any other value is a pointer to the window behind which this window should be placed. You should return the carry flag clear.

wCallDefProc, Operation 10

WCallDefProc is a generic call to the defProc that is defined by the defProc. With this call a window defProc can define many special functions.

The input to the defProc is:

param = pointer to the following parameter table:

dRequest	WORD	Requested operation number.
paramID	WORD	Parameter block type: $0000-$7FFF reserved by system ($0000 defined below). $8000-$FFFF reserved for custom defProcs.
newParam	BYTE[n]	New parameter field used by some operations.

The paramID field defines dRequest, which in turn defines newParam and the result of the wCallDefProc call. You can think of dRequest as the operation number passed to the defProc. Here is an example of how the paramID defines dRequest: if paramID is zero, dRequest 3 is defined as wSetPage (defined below); but if paramID is $8345 (or any number other than zero), dRequest 3 could be defined as something entirely different.

The following dRequest values are defined for wCallDefProc operations with a paramID of zero. Your defProc should check for handling only these codes. In the future, codes 34 and greater may be defined, and your defProc should know not to handle them.

wSetOrgMask	0	wGetInfoDraw	17
wSetMaxGrow	1	wGetOrigin	18
wSetScroll	2	wGetDataSize	19
wSetPage	3	wGetZoomRect	20
wSetInfoRefCon	4	wGetTitle	21
wSetInfoDraw	5	wGetColorTable	22
wSetOrigin	6	wGetFrameFlag	23
wSetDataSize	7	wGetInfoRect	24
wSetZoomRect	8	wGetDrawInfo	25
wSetTitle	9	wGetStartInfoDraw	26
wSetColorTable	10	wGetEndInfoDraw	27
wSetFrameFlag	11	wZoomWindow	28
wGetOrgMask	12	wStartDrawing	29
wGetMaxGrow	13	wStartMove	30
wGetScroll	14	wStartGrow	31
wGetPage	15	wNewSize	32
wGetInfoRefCon	16	wTask	33

wSetOrgMask 0

```
newParam  =  WORD - window's origin mask.
result    =  None.
```

Called when SetOriginMask is called.

wSetMaxGrow 1

```
newParam   =  WORD - maximum window height.
              WORD - maximum window width.
result     =  None.
```

Called when `SetMaxGrow` is called.

`wSetScroll          2`

```
newParam   =  WORD - number of pixels to scroll when arrow is
              selected.
result     =  None.
```

Called when `SetScroll` is called.

`wSetPage            3`

```
newParam   =  WORD - pixels to scroll when page region is selected.
result     =  None.
```

Called when `SetPage` is called.

`wSetInfoRefCon      4`

```
newParam   =  LONG - value passed to info bar draw routine
              (app’s use only).
result     =  None.
```

Called when `SetInfoRefCon` is called.

`wSetInfoDraw        5`

```
newParam   =  LONG - address of info bar draw routine.
result     =  None.
```

Called when `SetInfoDraw` is called.

`wSetOrigin          6`

```
newParam   =  WORD - flag, TRUE to scroll content.
              WORD - window’s Y origin.
              WORD - window’s X origin.
result     =  None.
```

Called when `SetContentOrigin` is called.

`wSetDataSize        7`

```
newParam   =  WORD - height of window’s data area.
              WORD - width of window’s data area.
result     =  None.
```

Called when `SetDataSize` is called.

`wSetZoomRect        8`

```
newParam   =  LONG - pointer to new zoom RECT.
result     =  None.
```

Called when `SetZoomRect` is called.

`wSetTitle           9`

```
newParam  =  LONG - pointer to new title.
result    =  None.
```

Called when `SetWTitle` is called.

```
wSetColorTable     10
      newParam  =  LONG - pointer to new color table.
      result    =  None.
```

Called when `SetFrameColor` is called.

```
wSetFrameFlag      11
      newParam  =  LONG - pointer to new zoom RECT.
      result    =  None.
```

Called when `SetWFrame` is called.

```
wGetOrgMask        12
      newParam  =  None.
      result    =  WORD - window's origin mask.

wGetMaxGrow        13
      newParam  =  None.
      result    =  Low word is window's maximum height when grown.
                   High word is window's maximum width when grown.
```

Called when `GetMaxGrow` is called.

```
wGetScroll         14
      newParam  =  None.
      result    =  Low word is number of pixels to scroll when arrow is selected.
```

Called when `GetScroll` is called.

```
wGetPage           15
      newParam  =  None.
      result    =  Low word is pixels to scroll when page region is selected.
```

Called when `GetPage` is called.

```
wGetInfoRefCon     16
      newParam  =  None.
      result    =  Value passed to info bar draw routine.
```

Called when `GetInfoRefCon` is called.

```
wGetInfoDraw       17
      newParam  =  None.
      result    =  Address of info bar draw routine.
```

Called when `GetInfoDraw` is called.

```
wGetOrigin         18
      newParam  =  None.
      result    =  Low word is content's Y origin.
                   High word is content's X origin.
```

Called when `GetContentOrigin` is called.

```
wGetDataSize       19
       newParam  =  None.
       result    =  Low word is window’s data height.
                    High word is window’s data width.
```

Called when `GetDataSize` is called.

```
wGetZoomRect       20
       newParam  =  None
       result    =  Pointer to window’s current zoom RECT.
```

Called when `GetZoomRect` is called.

```
wGetTitle          21
       newParam  =  None
       result    =  Pointer to window’s title.
```

Called when `SetWTitle` is called.

```
wGetColorTable            22
       newParam  =  None.
       result    =  Pointer to window’s color table.
```

Called when `SetFrameColor` is called.

```
wGetFrameFlag             23
       newParam  =  None.
       result    =  Low word is window’s wFrame field.
```

Called when `SetWFrame` is called.

```
wGetInfoRect       24
       newParam  =  LONG - pointer to place to store info bar’s enclosing RECT.
       result    =  None.
```

Called when `GetRectInfo` is called.

```
wGetDrawInfo       25
       newParam  =  None.
       result    =  None.
```

Called when `DrawInfoBar` is called.

```
wGetStartInfoDraw 26
       newParam  =  LONG - pointer to place to store info bar’s enclosing
                    RECT.
       result    =  None.
```

Called when `StartInfoDrawing` is called.

```
wGetEndInfoDraw           27
       newParam  =  None.
       result    =  None.
```

Called when `EndInfoDrawing` is called.

wZoomWindow 28

```
newParam  =  None.
result    =  None.
```

Called when `ZoomWindow` is called.

wStartDrawing 29

```
newParam  =  None.
result    =  None.
```

Called when `StartDrawing` is called.

wStartMove 30

```
newParam  =  WORD - new y position (global).
             WORD - x position (global).
result    =  Low word is new y position (global).
             High word is x position (global).
```

Called before `MoveWindow` moves a window.

wStartGrow 31

```
newParam  =  None.
result    =  None.
```

Called before `GrowWindow` tracks the growing of a window.

`wNewSize` 32

```
newParam  =  LONG - pointer to:
                          WORD - proposed new height.
                          WORD - proposed new width.
                          These two values can be changed.
result    =  Low word TRUE if only uncovered content should be drawn.
             FALSE if entire content should be redrawn.
```

Called by `SizeWindow` before it resizes a window. The new height and width can be changed by modifying the words pointed to by the pointer in `newParam`.

`wTask` 33

```
newParam  =  LONG - pointer to task record.
             WORD - result from FindWindow.
result    =  Low word is code returned by TaskMaster (zero if handled).
             High word is task performed.  Returned in TaskData if code is 0.
```

Called from `TaskMaster` when it cannot handle a task. If the user presses the mouse button over a window, `TaskMaster` will call `FindWindow` to find out what part of the window. `TaskMaster` will then handle the task if `FindWindow` returns `wInMenuBar` or bit 15 of the window pointer is set (system window). Otherwise, the result of `FindWindow` is passed to `wTask` to be handled or not.

If the `defProc` can handle the task it should do so and return zero in the low word of the result (which will be the result to the application returned from `TaskMaster`) and a code of the task performed in the high word of the result (which is returned to the application in its task record `TaskData` field). Fields in the task record may also be modified to return parameters to the application as this is the same record passed to `TaskMaster`.

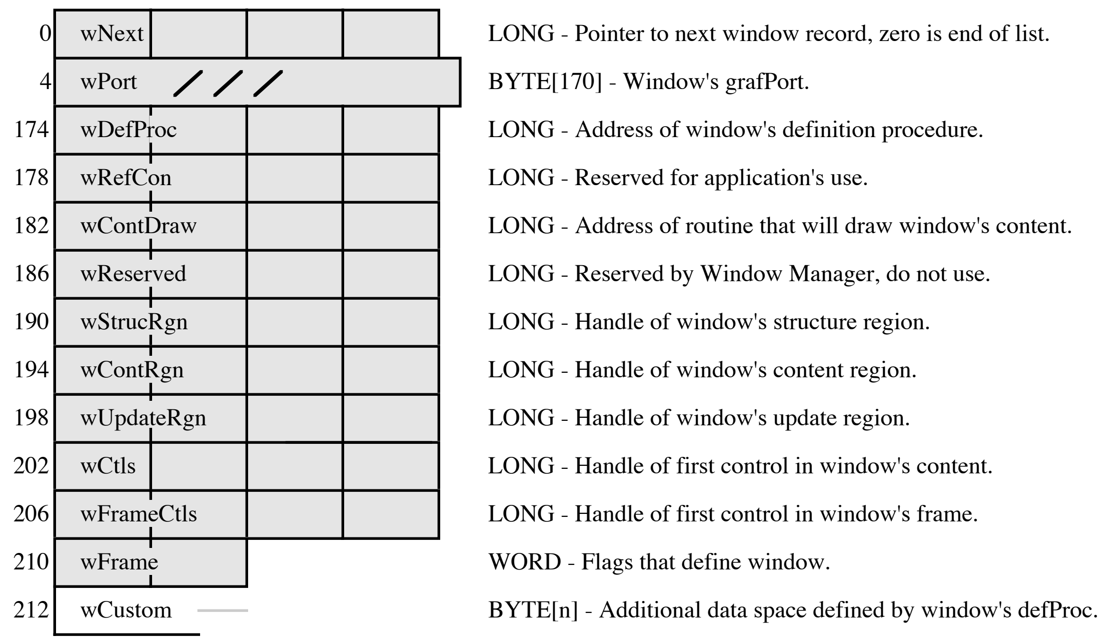
If the `defProc` cannot handle the task, it should return the result from `FindWindow` (the second field in `newParam`) in the low word of the result. The high word of the result is not used.

For example, the standard document window `defProc` handles the following results from `FindWindow` if the `taskMask` record allows.

`wInContent`	Brings the window to the top.
`wInDrag`	Calls `DragWindow`.
`wInGrow`	Brings the window to the top. If it is already on the top, it calls `GrowWindow` and `SizeWindow`.
`wInGoAway`	Calls `TrackGoAway`.
`wInZoom`	Calls `TrackZoom` and `ZoomWindow`.
`wInInfo`	Brings the window to the top.
`wInFrame`	Brings the window to the top. If it is already on the top, checks if it is on one of the window's scroll bars, tracks it, and scrolls the window's content as needed.

A custom window `defProc` can return any code (bit 15 is used for system windows) it wants when it is called to do a hit test. This code would be that returned by `FindWindow`, and the application would have to know about the code if it called `FindWindow` instead of `TaskMaster`. If `TaskMaster` is used, the code that `FindWindow` returns is passed back to your `defProc` with a `wCallDefProc` and `wTask`. The `defProc` could perform any task it wanted: change colors, eject a disk, run a spelling checker, or anything else.

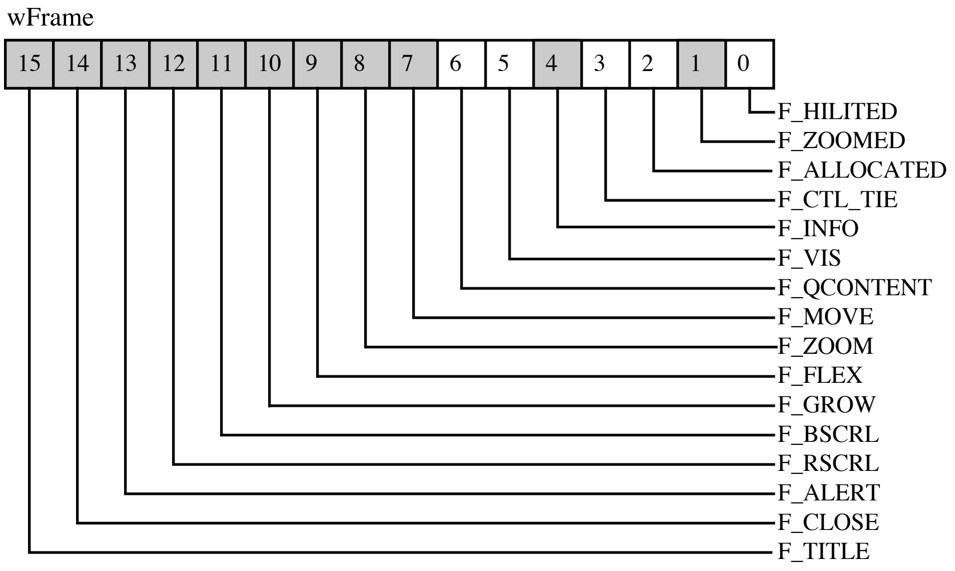
Window Record Definition



Offset	Field	Description
0	wNext	LONG - Pointer to next window record, zero is end of list.
4	wPort	BYTE[170] - Window's grafPort.
174	wDefProc	LONG - Address of window's definition procedure.
178	wRefCon	LONG - Reserved for application's use.
182	wContDraw	LONG - Address of routine that will draw window's content.
186	wReserved	LONG - Reserved by Window Manager, do not use.
190	wStrucRgn	LONG - Handle of window's structure region.
194	wContRgn	LONG - Handle of window's content region.
198	wUpdateRgn	LONG - Handle of window's update region.
202	wCtls	LONG - Handle of first control in window's content.
206	wFrameCtls	LONG - Handle of first control in window's frame.
210	wFrame	WORD - Flags that define window.
212	wCustom	BYTE[n] - Additional data space defined by window's defProc.

The changes use some vacant space under the window port and add the `wReserved` field to the record for future expansion.

In addition to defining the window record, the `wFrame` field needs to be further defined. In the diagram below the shaded bits are reserved for use by each window `defProc` (the values shown are those used by the standard document window `defProc`). Bits not shaded are reserved by the Window Manager and are applicable to all windows.



Further Reference

- *Apple IIGS Toolbox Reference*, Volume 1
- System Disk 4.0 Release Notes

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#43: Undocumented Feature of CalcMenuSize

Revised by:	Matt Deatherage	March 1991
Written by:	Dan Oliver	November 1988

This Technical Note documents that `CalcMenuSize` can accept a parameter of $FFFF to recalculate menus with uninitialized heights and widths.
Changes since November 1988: This Note is now obsolete.

This Note formerly described how `CalcMenuSize` behaves when menu widths and heights are stored as $0000 or $FFFF. This behavior is now documented in Volume 3 of the *Apple IIGS Toolbox Reference* on page 37-3.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 3

Apple IIGS
#44: GetPenState and SetPenState Record Error

Revised by: Matt Deatherage March 1991
Written by: Keith Rollin November 1988

This Technical Note corrects an error in the record used for `GetPenState` and `SetPenState`.
Changes since November 1988: This note is now obsolete.

This Note formerly described an error in the pen state record in Volume 1 of the *Apple IIGS Toolbox Reference*. This error is corrected on page 43-2 of Volume 3 of the *Toolbox Reference*.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 3

Apple IIGS
#45: Parameters for GetFrameColor

Revised by: Matt Deatherage September 1989
Written by: Dan Oliver November 1988

This Technical Note formerly attempted to correct the description of the parameters passed to and returned from the routine `GetFrameColor` in the Window Manager chapter of the *Apple IIGS Toolbox Reference*. This call works as documented since System Software 3.2; therefore, former versions of this Note were incorrect.
Changes since November 1988: Corrected our error. Sorry for any inconvenience.

This Note formerly stated the following: "The *Apple IIGS Toolbox Reference,* Volume 2 incorrectly describes the parameters passed to and returned from `GetFrameColor` on page 25–57."

However, this is incorrect. Beginning with System Software 3.2, `GetFrameColor` works as documented in the *Apple IIGS Toolbox Reference,* Volume 2. Prior to System Software 3.2, the call did not work at all. We apologize for any inconvenience this confusion may have caused.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#46: DrawPicture Data Format

Written by: Jeff Erickson & Keith Rollin November 1988

This Technical Note describes the internal format of the QuickDraw II picture data structure.

This Technical Note presents the internal format of the QuickDraw II picture data structure for informational purposes only. You should **not** use this information to write your own bottleneck procedures; the only routines which should create and read `PICT` format files are those provided in QuickDraw II. If we added new objects to the picture definition, your program would not operate on new pictures. This Note documents this information for **debugging purposes only**.

Picture Data Structure Definition

Pictures are stored in memory in the following format:

They begin with a `WORD` which indicates the mode of the port which was used to record when the picture was created. This information is useful when the picture is played back, possibly in a different graphics mode.

Following the `WORD` is a `RECT` which indicates the frame of the picture and is used for scaling when you redraw the picture. Following the `RECT` is the version number of this `PICT` format, then a series of word-sized opcodes which describe the sequences of QuickDraw II commands that were used to create the picture.

Name	**Description**	**Size (bytes)**
`pictSCB`	picture’s scan line control byte	2 (high byte = 0)
`picFrame`	picture’s boundary rectangle	8
`version`	picture version	2 (Currently $8211)
`opcode`	operation code	2
`<data>`	operation data	variable, depending on opcode
`:`		
`opcode`	operation code	2
`<data>`	operation data	variable, depending on opcode

Opcodes

As mentioned above, pictures are described by a series of opcodes which are used to record the QuickDraw II commands that created the picture. These opcodes are two bytes long and are usually followed by a number of parameters.

All currently defined opcodes and their parameters are listed below. Any opcodes not listed here are reserved.

Opcode	Name	Description	Parm Bytes	Parameter Description
$0000	`NOP`	no operation	0	none
$0001	`ClipRgn`	clip to a region	[region size]	region
$0002	`BkPat`	background pattern	32	background pattern (8x8 pixels)
$0003	`TxFont`	text font	4	Font Manager font ID (long)
$0004	`TxFace`	text face	2	text face (word)
$0005	`TxMode`	text mode	2	text mode (word)
$0006	`SpExtra`	space extra	4	space extra (fixed)
$0007	`PnSize`	pen size	4	pen size (point)
$0008	`PnMode`	pen mode	2	pen mode (word)
$0009	`PnPat`	pen pattern	32	pen pattern (8x8 pixels)
$000A	`FillPat`	fill pattern	32	fill pattern (8x8 pixels)
$000B	`OvSize`	oval size	4	oval size (point)
$000C	`Origin`	origin	4	origin (point)
$000D	`TxSize`	text size	2	text size (word)
$000E	`FGColor`	foreground color	2	color (word)
$000F	`BGColor`	background color	2	color (word)
$XX11	`Version`	version	0	none: high byte = version (currently $82)
$0012	`ChExtra`	character extra	4	char. extra (fixed)
$0013	`PnMask`	pen mask	8	mask (8 bytes)
$0014	`ArcRot`	arc rot	2	Reserved (related to things drawn w/patterns). (word)
$0015	`FontFlags`	font flags	2	font flags (word)
$0020	`Line`	line	8	`pnLoc` (point), `newPt` (point)
$0021	`LineFrom`	line from pen loc.	4	`newPt` (point)
$0022	`ShortLine`	short line	6	`pnLoc` (point), `dv`, `dh` (signed bytes)
$0023	`ShortLFrom`	ditto from pen loc	2	`dv`, `dh` (signed bytes)
$0028	`LongText`	long text	5+text	`txLoc` (point), `count` (byte), text
$0029	`DHText`	hor. offset text	2+text	`dh` (unsigned byte), `count` (byte), text
$002A	`DVText`	vert. offset text	2+text	`dv` (unsigned byte), `count` (byte), text
$002B	`DHDVText`	offset text	3+text	`dv`, `dh` (unsigned bytes), `count` (byte), text

$002C	`RealLongText` very long text	6+text	`txLoc` (point), `count` (word), text

Opcodes between $0030 and $008C are a combination of a graphic verb and a graphic object, as listed below (where “V” stands for the graphic verb, and “X” is a stands for the graphic object). For example, $0069 means `PaintSameArc`, and is followed by two one-word parameters.

Graphic Verbs:

$00X0	`Frame…`	frame something	[Specific to object type: see below.]
$00X1	`Paint…`	paint something	
$00X2	`Erase…`	erase something	
$00X3	`Invert…`	invert something	
$00X4	`Fill…`	fill something	
$00XV+8	`…Same…`	draw same thing somehow	[See below; underlined parms do not appear.]

Graphic Objects:

$003V	`…Rect`	draw a rectangle somehow	8 (0 if – SameRect) rect (2 points)
$004V	`…RRect`	draw a round rect somehow	8 (0) rect (2 points)
$005V	`…Oval`	draw an oval somehow	8 (0) rect (2 points)
$006V	`…Arc`	draw an arc somehow	12 (4) rect (2 points), start, arc angle (words)
$007V	`…Poly`	draw a polygon somehow	[polygon size] (0) polygon
$008V	`…Rgn`	draw a region somehow	[region size] (0) region
$0090	`BitsRect`	copybits, rect clipped	variable† (see below, but without `maskRgn`)
$0091	`BitsRgn`	copybits, rgn clipped	variable† (see below)
$00A1	`LongComment`	long comment	4+data kind (word), size (word), data

†Bits… data:

`origSCB`	original scan line control byte	2	SCB (word — high byte = 0)
`BWvsColor`	black and white vs. color	2	reserved (word)
`width`	width of pixel image in bytes	2	width (word)
`boundsRect`	bounds rectangle	8	rect (2 points)
`srcRect`	source rectangle	8	rect (2 points)
`destRect`	destination rectangle	8	rect (2 points)
`mode`	transfer mode	2	pen mode (word)
`maskRgn`	mask region (BitsRgn ONLY!)	[region size]	region
`pixData`	pixel image	[pixdata size]	width*(bounds.bottom-bounds.top)

Differences Between IIGS Pictures and Macintosh Pictures

1. QuickDraw II pictures are modeled after `PICT2` on the Macintosh, which use two bytes for its opcodes and data (the exception to this is the $11 (version) opcode, which is followed by a one-byte parameter). Macintosh `PICT` 1.0 formats, which use one-byte opcodes, would have to undergo extensive modifications to be displayed on the IIGS.

2. There is no `EndOfPicture` opcode on the IIGS as there is on the Macintosh. Also, the first word of the picture is a `pictSCB`, not the length of the picture. The picture size is determined solely by the size of the handle on the IIGS. There is also no picture header on the IIGS as on the Macintosh.
3. The number sex of the Macintosh is opposite that of the Apple IIGS. The Macintosh stores the high bytes of words and long words first, whereas the IIGS stores the low byte first.
4. The following Macintosh picture opcodes are not available on the IIGS: `txRatio`, `PackBitsRect`, `PackBitsRgn`, `shortComment`, `EndOfPicture`.
5. QuickDraw II defines the following opcodes that the Macintosh does not: `ChExtra` ($12), `PnMask` ($13), `ArcRot` ($14), `FontFlags` ($15), and `RealLongText` ($2C).

Notes on the Interpretation of IIGS Pictures

- The state of the pen, the clip region, various patterns and colors, and the origin of the current port is saved before a picture is drawn, and restored afterwards. The current port is set up in a default state equivalent to that of a newly created port just before drawing begins. Picture opcodes act just like their QuickDraw II tool counterparts, with a few exceptions.
- Two pen locations are tracked as the picture is drawn, one for lines and one for text. Thus, `LineFrom` always draws from the end of the last line, regardless of any intermediate text opcodes.
- Text calls do not change the position of the "text pen," as do normal QuickDraw II text calls. Thus, if a picture contains two lines of text, the second one directly below the first, the second will be stored using a `DVtext` opcode.
- `DrawPicture` performs considerable setup before it draws pictures. Among other things, it calls `InstallFont`, which is a Font Manager call. If you are going to support pictures in your application, you should load and start the Font Manager.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2

Apple IIGS
#47: What SetDataSize Does

Written by: Keith Rollin November 1988

This Technical Note clears up any ambiguity in the description of the `SetDataSize` call.

The Apple IIGS supports windows that contain scroll bars in their frames. These scroll bars are handled by `TaskMaster` and differ from Macintosh scroll bars in that the size of the "thumb" or "elevator" is used to indicate the size of the visible area of the document in relation to the total size of the document (the "data size"). Initially, the visible size and the data size are defined by the parameter list passed to `NewWindow`; however, either of these can be changed by `SizeWindow` and `SetDataSize`, respectively.

`SetDataSize` is used to not only change the range of scrolling allowed, but also to redraw the size of the thumb to reflect the fact that the data size has changed with respect to the visible area. However, page 25-97 of the *Apple IIGS Toolbox Reference* contains the following description of `SetDataSize`:

"Sets the height and width of the data area of a specified window. Setting these values will not change the scroll bars or generate update events."

When the manual states that `SetDataSize` "will not change the scroll bars," it is referring to the location, or value, of the thumb. Assume a situation where you have a word processor that scrolls the page using `TaskMaster` scroll bars. If you delete a range of text, you would also shorten the entire size of the document. Calling `SetDataSize` to reflect that would indeed change the size of the thumb, but it would not change its location. If you were already scrolled to the bottom of the document when you called `SetDataSize`, the thumb would become larger (to reflect the fact the the total data size became smaller with respect to the visible data size) and overwrite the down arrow of the scroll bar. To prevent this situation from occurring, you should also change the origin of the window with `SetContentOrigin` before calling `SetDataSize`.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2

Apple IIGS
#48: All About AlertWindow

Revised by: Matt Deatherage & Dave Lyons November 1990
Written by: Dan Oliver & Keith Rollin November 1988

This Technical Note formerly documented the features and behavior of `AlertWindow`.
Changes since July 1989: The information on `AlertWindow` formerly found in this Note has been updated and is now included in the *Apple IIGS Toolbox Reference*, Volume 3.

The information on AlertWindow formerly found in this Note has been updated and is now included in the *Apple IIGS Toolbox Reference*, Volume 3. This Window Manager call was first introduced in System Software 3.2.

Apple IIGS
#49: Rebooting (Really)

Revised by: Matt Deatherage — January 1989
Written by: Matt Deatherage & Jim Merritt — November 1988

This Technical Note discusses rebooting the Apple IIGS from software.
Changed since November 1988: Corrected two assembly-language instructions in the `FROMNATV` routine in the example code.

In days gone by, many Apple II applications had a Quit menu option. Unfortunately, a large number of these simply rebooted the machine. Today, this is far from desirable. Even with the advantages of GS/OS-reduced booting time (around 34 seconds with an Apple 3.5 Drive), waiting for the operating system to reload, as well as wiping out any ongoing tasks by desk accessories (such as an alarm clock) makes the standard ProDOS 8 or GS/OS `QUIT` call much more attractive.

However, there are still instances where an application may wish to require the user to reboot. A common example might be a game. The game might use GS/OS in a completely standard way, but if you `QUIT` from the program GS/OS booted into, you will be returned to the same program. Since most applications will boot into the Finder, this is not a widespread problem. However, the Finder must also provide the reboot option, and alternate program selector applications may wish to provide this functionality as well.

The Easy Way

GS/OS provides a mechanism for rebooting with the `OSShutdown` call. This call, documented in *GS/OS Reference*, Volume 1, will either reboot the system (after first shutting down all loaded and generated drivers and closing all open sessions) or will shut down everything and present a dialog box which states, “You may now power down your Apple IIGS safely.” A Restart button is provided which allows the user to reboot without pressing Control-Open Apple-Reset .

Note: When using System Disk 4.0, if the Window Manager is active when you issue the `OSShutdown` call, there must be at least one open window; it need not be visible, but it must be open. This will be fixed in the next revision of GS/OS.

The `OSShutdown` call also provides a way to resize the internal RAM disk (named /RAM5 by default). Most programs have absolutely no need to use this mechanism, and should avoid it whenever possible. A notable exception would be a third-party RAM disk utility which uses a battery backup, which may need to make changes which require resizing the RAM disk. Of course, such a utility should ask the user to ensure that erasing the RAM disk content is acceptable. Resizing the RAM disk is only possible when using the `OSShutdown` call; any other method you may be using to accomplish this function from software will break in the future.

If you are using GS/OS, you should **always** use `OSShutdown`. You must not reboot the system in any other fashion. The `OSShutdown` mechanism provides a convenient and supported way to restart or shut down the system. Doing it another way can easily cause a loss of data.

The Hard Way

Programs not using GS/OS have a little more work to do. The supported non-GS/OS method of rebooting is similar to the method used on 8-bit machines: change the value of `POWERUP` ($00/03F4) and do a long jump to `RESET` ($FA62). However, there are a few catches:

1. The jump must be made in emulation mode.
2. Interrupts must be disabled.
3. The data bank register must be set to zero.
4. The direct page must be zero.
5. ROM firmware must be visible in the memory map.
6. Internal interrupt sources (such as the ones for AppleTalk) must be shut down.

Simply disabling interrupts without shutting down AppleTalk interrupt sources inside the system will cause the system to hang when the jump to `RESET` is made. Turning off these internal interrupt sources is accomplished by changing softswitch values at $C039 (`SCCAREG`), $C041 (`INTEN`), and $C047 (`CLRVBLINT`).

The following code example demonstrates the correct method:

```
POWRUP        equ    $0003F4             ;the power-up byte in bank zero
STATEREG      equ    $C068               ;ROM/RAM state register
CLRVBLINT     equ    $C047               ;clear VBL interrupt flags register
INTEN         equ    $C041               ;interrupt enable register
SCCAREG       equ    $C039               ;SCC register
RESET         equ    $00FA62             ;ROM reset entry point
;
FROMNATV      anop                       ;enter here from native mode
              sei                        ;disable interrupts
              pea    0
              pea    0                   ;push four zero bytes on the stack
              plb                        ;pull data bank register
              plb                        ;(twice to balance the stack)
              pld                        ;pull 16-bit data bank register
              sec
              xce                        ;go into emulation mode
              longa  off
              longi  off
FROMEMUL      anop                       ;enter here from emulation mode
              sei                        ;disable interrupts for people entering here
              dec    POWRUP              ;invalidate the power up byte
              lda    #$0C                ;ROM parameters
              sta    STATEREG            ;swap in the ROM and everything else out
              stz    CLRVBLINT           ;clear VBL interrupts
              stz    INTEN               ;turn off internal interrupt sources
              lda    #$09
              sta    SCCAREG             ;shut down SCC interrupt sources
              lda    #$C0
              sta    SCCAREG
              jml    RESET               ;and off we go into the wild blue yonder
```

These methods of restarting the system are presented for those applications that absolutely must do so. Rebooting is not a suggested way of ending an application and the techniques described in this Note should be used with **extreme** caution.

Further Reference

- *Apple IIGS Firmware Reference*
- *GS/OS Reference*, Volume 1